Workflow Function Reference

Levels Beyond
2710 Walnut Street | Denver, CO 80205 | (303) 495-2424
www.levelsbeyond.com

reachENGINE

Table of Contents

Introduction

This guide provides a list and explanation of the standard functions available for all expressions in Reach Engine. Expressions should always be written in the form:

```
${...#functionname(arg1.. argn)...}
```

Functions are available for the following:

- Files
- System properties and utilities
- Dates
- Metadata
- Timecodes
- Assets
- Nimbus only

Function Name	Argument(s)	Outcome
#absPath	*(fileOrName)* A file that specifies the target file. **Example:** `#absPath(mickey.mov)`	Returns the absolute path of the specified file.
#baseFilename	*(fileOrName)* A file or string that specifies the target file or filename. **Example:** `#baseFilename(mickey)`	Returns the name of fileOrName without its extension. This function is useful to copy a file with a different extension.
#ls	*(directoryOrPath, includeFiles, includeDirectories, extensionFilter)* • *directoryOrPath:* A file or string that specifies the target directory or path. • *includeFiles:* A Boolean that determines whether files should be returned from the function. • *includeDirectories:* A Boolean that determines whether child directories should be returned from the function. • *extensionFilter:* A string (can be multiple) that provides an optional list of extension strings that restricts the included files to the ones matching one of the extensions given. Directory inclusion is unaffected by this argument. **Example:** `#ls(path, true, true, mov)`	Returns the contents of the specified directory. Result can be filtered by files (include true/false), directories (include true/false), and specified extensions.

#extension	*(fileOrName)* A file or string that specifies the target file or filename. **Example:** `#extension(mov)`	Returns the extension of fileOrName.
#fileExists	*(fileOrPath)* A file or string that specifies the target file or full path to a file. **Example:** `#extension(mov)`	Returns true if the file exists or false if it does not.
#filename	*(fileOrPath)* A file or string that specifies the target file or full path to a file. **Example:** `#filename(mickey.mov)`	Returns just the file name of the given file or path. The leading file path is removed.
#filepath	*(fileOrPath)* A file or string that specifies the target file or full path to a file. **Example:** `#filepath(C:/database/movies/mickey.mov)`	Returns just the path portion of the given file or path. Filenames are removed.

#mediaType	*(input)*
	input input object: A file, path, or asset
	Example
	Expression:
	`#mediaType(asset)`
	Context Data Defs:
	`<contextDataDef name="output" dataType="String" multiple="false"/>`
		Returns the media type of the given file or file wrapper (asset).

#walkDir	*(fileOrPath, extensions)*
	• *fileOrPath:* A file or string that specifies the target directory or full path to a directory
	• *Extensions:* A collection of string file extensions, which may be represented by a multiple=true string type data def
	Example
	Expressions:
	`#walkDir(file, extensions)`
	`#walkDir(path, extensions)`
	Context Data Defs:
	`<!–input -->`
	`<contextDataDef name="file" dataType="File" multiple="false"/>`
	`<contextDataDef name="path" dataType="String" multiple="false"/>`
	`<contextDataDef name="extensions" dataType="String" multiple="true"/>`
	`<!–output -->`
	`<contextDataDef name="files" dataType="File" multiple="true"/>`
		Returns a collection of files from the given directory, matching the given extension filters.

File Functions

Function Name	Argument(s)	Outcome

#camelCase	*(string)*
	This function will camel case an input string, splitting on space, hyphen, and underscore.
	Example
	Expression:
	`#camelCase("camel case-this_string")`
	Output:
	`camelCaseThisString`
		Returns a camel-cased version of the input string.

#cleanList	*(list, dedupe, removeNulls)*
	• *list:* Input list of objects
	• *dedupe:* Set to "true" to remove duplicates.
	• *removeNulls:* Set to "true" to remove null values.
	Example
	Expression:
	`#cleanList(list, true, true)`
	Context Data Defs:
	`<contextDataDef name="list" dataType="Data Object" mutliple="true"/>`
	`<contextDataDef name="output" dataType="Data Object" mutliple="true"/>`
		Cleans the given list by optionally removing duplicate and null values.
		Returns the updated list.

#convertToJSON	*(string)*	Converts a valid JSON string to JSON Object
#conversionTemplates	*(converterName, prefix)* • *converterName:* optional name of the converter • *prefix:* optional prefix of the transcode template **Example** Expression: `#conversionTemplates('FFmpeg', 'import')`	Returns a string collection of conversion templates available in Reach Engine
#decodeURL	*(string)*	Runs java URL decoder on the string, returns decoded string.
#getChildFiles	*(parentDir, includeFiles, includeDirectories)* • *parentDirectory:* Directory to scrub • *includeFiles:* Boolean, include Files in your result • includeDirectories: Boolean, include Directories in your result	Scrubs and returns a list of objects of what is in that directory.

#joinElements	*(collection, separator)*
	• *collection:* A collection of strings or objects to be joined as a string
	• *separator:* The custom item separator
	Example
	Expression:
	`#joinElements(strings, ', ')`
	Context Data Defs:
	`<contextDataDef name="strings" dataType="String" multiple="true"/>`
	`<contextDataDef name="output" dataType="String" mutliple="false"/>`
		Returns a string representation of all elements in the given collection, delimited by the given separator.

#randomNumber	*(min, max)*
	• *min:* The minimum number that can be returned
	• *max:* The maximum number that can be returned
	Example 1
	Expression:
	`#randomNumber(1, 3)`
	Context Data Defs:
	`<!-- random number between 1 and 3, inclusive -->`
	`<contextDataDef name="output" dataType="Integer" multiple="false"/>`
	Example 2
	Expression:
	`#randomNumber(1.5, 3.7)`
	Context Data Defs:
	`<!-- random number between 1.5 and 3.7, inclusive -->`
	`<contextDataDef name="output" dataType="Double" multiple="false"/>`
		Returns a random number within the given number range.

#mapPath	*(path, endpoint)*
	• *path:* Input path
	• *endpoint:* Mapped path mapping endpoint
	Example
	Expression:
	`#mapPath('/Volumes/reachengine/media/video.mov', 'global')`
	Context Data Defs:
	`<!-- /reachengine/media/video.mov -->`
	`<contextDataDef name="output" dataType="String" multiple="false"/>`
		Exposes path mapping functionality to Reach Engine expressions.
		Maps a given path to the local Reach Engine equivalent.

#splitString	*(string, delimiter, trimResults, removeEmpty)*
	• *string:* Source string
	• *delimiter:* The substring for splitting
	• *trimResults:* Whether or not to trim whitespace from the resulting substrings
	• *removeEmpty:* Whether or not to remove empty strings from the result set
	Example
	Expression:
	`#splitString("abc", "bc", true, true)`
	Context Data Defs:
	`<contextDataDef name="output"`
	`dataType="String" multiple="true"/>`
		Returns a collection of strings which have been derived from the input string.
		Example
		`#splitString("a b c", "b", true, true)`
		`#splitString("a b c", "b c", true, true)`
		`#splitString("a b c", "b c", true, false)`
		`#splitString("a b c", "b c", false, true)`

#startsWith	*(input, test)*
	• *input:* The input string to test
	• *test:* The substring to test
	Example
	Expression:
	`#startsWith('abcde', 'ab')`
	Context Data Defs:
	`<contextDataDef name="output"`
	`dataType="Boolean" multiple="false"/>`
		Returns "true" if the input string starts with the test string.
		Expression:
		`#startsWith('abcde', 'ab')`
		`#startsWith('abcde', 'abcde')`
		Returns "false" if the input string does not start with the test string.
		Expression:
		`#startsWith('abcde', 'cde')`
		`#startsWith('abcde', 'abd')`
		`#startsWith('abcde', 'abcdef')`

#sysconfig	*(propertyName)* A string that specifies the system config property to read. **Example:** `#sysconfig(workflow.import)`	Returns the configured value of the given property name. This function allows you to reference any default or configured property value in *local.reach-engine.properties*.
#uuid()	*()* This function takes no arguments. **Example:** `${#uuid().toString()}`	Returns a 128-bit UUID string, like 49fc3e9c-1103-410e-bdcd-118e04ac916b. This function is useful for creating unique file names. All Java String methods are available from the result.

#xpath	*(xml, xpath, multipleResults)*
	• *xml:* The XML file against which you want to evaluate
	• *xpath:* The XPath query to apply
	• *multipleResults:* Determines whether to return all matching values or just the first match
	Example:
	`#xpath(xmlObject,'//resources/asset/@src',true)`
	With Context Data Defs:
	`<contextDataDef name="XmlFile" dataType="File" userInput="true"/>`
	`<contextDataDef name="XmlObject" dataType="XML" defaultDataExpression="${XmlFile}"/>`
	And an XML file like:
	`<fcpxml>`
	`<project>`
	`<resources>`
	`<asset id="r6" name="myVideo" src="file://localhost/Volumes/Media/myFile.mov">`
	`</asset>`
	`<asset id="r7" name="myVideo" src="file://localhost/Volumes/Media/myFile2.mov">`
	`</asset>`
		Returns the result of the XPath evaluation. If multipleResults is *true*, the returned value is a collection of XML objects, otherwise it is a single XML object.
		Result of the XPath Argument:
		file://localhost/Volumes/Media/myFile.mov
		file://localhost/Volumes/Media/myFile2.mov
		Refer to the following samples for testing this functionality:
		Sample XML File to Test With
		Sample Workflow

System Properties and Utilities

Function Name	Argument(s)	Outcome
#newDate	*()* This function takes no arguments.	Returns a new date value, that can be stored as a date object like **myDate**, equal to the current system time. **Sample Result:** Mon Oct 07 09:06:31 PDT 2013

Note:
In the examples below, the variable **myDate** is used. This is usually created with a contextDataDef using the function above similar to:
`<contextDataDef name="myDate" dataType="Date" defaultDataExpression="${#newDate()}">`
`</contextDataDef>`
		If used in a string, would come out similar to above.
		Sample Result:
		Mon Oct 07 09:06:31 PDT 2013

#addDays	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of days to add. Negative numbers will subtract. **Example:** `#addDays(myDate, -1)`	Returns the entered date adjusted by the number of days specified.
#addHours	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of hours to add. Negative numbers will subtract. **Example:** `#addHours(myDate, 3)`	Returns the entered date adjusted by the number of hours specified.
#addMilliseconds	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of milliseconds to add. Negative numbers will subtract.	Returns the entered date adjusted by the number of milliseconds specified.
#addMinutes	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of minutes to add. Negative numbers will subtract.	Returns the entered date adjusted by the number of minutes specified.
#addMonths	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of months to add. Negative numbers will subtract.	Returns the entered date adjusted by the number of months specified.
#addSeconds	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of seconds to add. Negative numbers will subtract.	Returns the entered date adjusted by the number of seconds specified.
#addWeeks	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of weeks to add. Negative numbers will subtract.	Returns the entered date adjusted by the number of weeks specified.

#addYears	*(date, amount)* • *date:* A date or date/time value that specifies the adjustment time. • *amount:* An integer that specifies the number of years to add. Negative numbers will subtract.	Returns the entered date adjusted by the number of years specified.
#formatDate	*(pattern, date)* • *pattern:* A string that specifies the date formatting pattern to use or "w3c" to generate a W3C date/time (required for Reach Engine timestamp parsing). • *date:* Specifies the date to format. **Example:** `#formatDate('DD/MM/YYY', myDate)`	Returns a string representation of the given date using the specified date pattern. **Sample Result:** 07/10/2013
#getDay	*(date)* The date or date/time to read. **Example:** `#getDay(myDate)`	Returns the date's day of the month (1-31). **Sample Result:** 7
#getHour	(date) The date or date/time to read. **Example:** `#getHour(myDate)`	Returns the date's hour (0-23). **Sample Result:** 9
#getMilliseconds	*(date)* The date or date/time to read.	Returns the date's milliseconds (0-999).
#getMinute	*(date)* The date or date/time to read.	Returns the date's minute (0-59).
#getMonth	*(date)* The date or date/time to read.	Returns the date's month (1-12).
#getSecond	*(date)* The date or date/time to read.	Returns the date's second (0-59)
#getYear	*(date)* The date or date/time to read.	Returns the date's year.

#parseDate	*(pattern, dateString)*
	• *pattern:* A string that specifies the date formatting pattern to use or "w3c" to generate a W3C date/time (required for Reach Engine timestamp parsing).
	• *dateString:* Specifies the formatted date.
	Example
	Expression:
	`#parseDate('DD/MM/YYY', dateString)`
	Context Data Defs:
	`<contextDataDef name="output" dataType="Date" multiple="false"/>`
		Returns a date of the given date string representation using the specified date pattern.
		Sample Result:
		07/10/2013

#setDate	*(date, years, months, days, hours, minutes, seconds, milliseconds)*
	• *date:* The date or date/time to modify
	• *years:* An integer value for the year that the date should be set to
	• *months:* An integer value for the month that the date should be set to
	• *days:* An integer value for the day of the month that the date should be set to
	• *hours:* An integer value for the hour of the day that the date should be set to
	• *minutes:* An integer value for the minutes of the hours that the date should be set to
	• *seconds:* An integer value for the seconds of the minute that the date should be set to
	• *milliseconds:* An integer value for the milliseconds of the second that the date should be set to
	Example:
	Expression
	`#setDate(myDate)`
	Context Data Defs:
	`<contextDataDef name="myDate" dataType="Date"</contextDataDef>`
		Returns the given date but with changes to the parts specified. Not all date value parameters are required; you can stop at any time.
		For example, if you want to just change the month of the given date, you could combine some functions.
		Example:
		`#setDate (date, #getYear (date), 9)`
		This argument would set the date variable's month to September but leave everything else as-is.

Date Functions

Function Name	Argument(s)	Outcome

#picklistItemExists	*(propertyName, label)*
	• *propertyName:* The name of the picklist property
	• *label:* The label for which to search
	Example
	Expression:
	`#picklistItemExists('workflowStatus', 'In Progress')`
	Context Data Defs:
	`<contextDataDef name="output" dataType="Boolean" multiple="false"/>`
		Determines if a metadata property picklist item exists. Use #picklistItemExists to manage the picklist population within a workflow.
		• Returns "true" if the label already exists for the given metadata field.
		• Returns "false" otherwise.
		• Returns an "IllegalArgumentException" error if there is no property named *propertyName*, or if that property does not have a picklist.

#picklistLabel	*(property, value)*
	• *property:* The picklist property or property path
	• *value:* The value to be converted
	Example
	Expression:
	`#picklistLabel('workflowStatus', 1)`
	`#picklistLabel('Project.nleType', 1)`
	Context Data Defs:
	`<contextDataDef name="output"`
	`dataType="String"`
	`multiple="false"/>`
		For a picklist property, converts a database value to the corresponding user-readable picklist label.
		• Returns the picklist label for the given property and value.
		• This function should be used for single-select picklist property fields.
		• In the absence of a class name, this function assumes the property to be a metadata property.
		• If the property does not exist, or if the property does not have a picklist defined, the return value will be equal to the given value.

#picklistValue	*(property, label)*	returns the pick list value for a given label

#picklistLabels	*(property, value, separator)*
	• *property:* The picklist property or property path
	• *value:* The value to be converted
	• *separator:* The value separator
	Example 1
	Expression:
	`#picklistLabels('year', value, null)`
	Context Data Defs:
	`<contextDataDef name="value" dataType="String" multiple="false"/>`
	`<!-- 2013 -->`
	`<contextDataDef name="output" dataType="String" multiple="false"/>`
	Example 2
	Expression:
	`#picklistLabels('years', value, '|')`
	Context Data Defs:
	`<contextDataDef name="values" dataType="String" multiple="true"/>`
	`<!-- 2012|2013|2014 -->`
	`<contextDataDef name="output" dataType="String" multiple="false"/>`
		For a picklist property, converts a database value or collection of values to the corresponding user-readable picklist label(s).
		• Returns a string representing a single value or set of values delimited by the given separator.
		• This function should be used for multi-select picklist property fields.
		• In the absence of a class name, this function assumes the property to be a metadata property.
		• If the property does not exist, or if the property does not have a picklist defined, the return value will be equal to the given value.

Metadata Functions

Function Name	Argument(s)	Outcome

#formatDuration	*(pattern, duration)*
	• *pattern:* The format pattern
	See DurationFormatUtils
	• *duration:* The duration to format. Format is performed in milliseconds.
	○ Floating point numbers (double, float) are assumed to be in seconds.
	○ Other numbers are assumed to be in milliseconds.
	Example
	Expression:
	`#formatDate('H:mm:ss.SSS', duration)`
	Context Data Defs:
	`<contextDataDef name="duration" dataType="Double" multiple="false" />`
	`<contextDataDef name="output" dataType="String" multiple="false" />`
		Throws an IllegalArgumentException error for an invalid number format.
		Returns a formatted string.

#formatNumber	*(pattern, number)*
	• *pattern:* The format pattern
	• *number:* The number to format
	Example
	Expression:
	`#formatNumber('#,###.####', number)`
	Context Data Defs:
	`<contextDataDef name="number" dataType="Double" multiple="false" />`
	`<contextDataDef name="output" dataType="String" multiple="false" />`
		Returns a formatted number string.
		Throws a NumberFormatException error for an invalid number.

#framerate	*(sequenceTimebase, isNtsc)*
	• *sequenceTimebase:* Timebase of the media
	• *isNtsc:* "true" if the media is NTSC standard
	Example
	Expression:
	`#framerate(30, true)`
	Context Data Defs:
	`<contextDataDef name="output" dataType="Double" multiple="false" />`
		Returns the Framerate enumeration, whose to-string can be converted to a double.

#timecode	*(startTime, offset, framerate, isDropframe)* • *startTime:* A double value that specifies the base time to evaluate, expressed in fractions of seconds. • *offset:* A double value that specifies the optional offset value to apply to startTime. Useful for working with clips. Dropframe is accounted for in the offset calculation. Default to 0.0 if you do not need an offset value. • *framerate:* The framerate to calculate the timecode against • *isDropframe:* A Boolean that determines whether the timecode should be in dropframe format. **Example:** `#timecode(2.4, 0.3, 24, true)`	Returns the SMPTE timecode of the given arguments. Dropframe timecodes are formatted as *HH:MM:SS;FF*. Non-dropframe timecodes are formatted *HH:MM:SS:FF*.

#timecodeFormat	*(timecodeString)*
	• *timecodeString:* A string representation of a timecode
	Example
	Expression:
	`#timecodeFormat('00:10:30;00')`
	Context Data Defs:
	`<contextDataDef name="output" dataType="String" multiple="false"/>`
		Returns the TimecodeFormat enumeration, whose to-string can be one of the following:
		• DROP_FRAME
		• NON_DROP_FRAME

#timecodeInSeconds	*(frameCount, frameRate, format)*
	• *frameCount:* Offset as a number of frames
	• *frameRate:* Timecode frame rate
	• *format:* TimecodeFormat enumeration
	See #timecodeFormat
	Example
	Expression:
	`#timecodeInSeconds(999, 29.97, #timecodeFormat('00:10:30;00'))`
	Context Data Def:
	`<contextDataDef name="output" dataType="Double" multiple="false"/>`
		Returns a double-value representation of duration in seconds.

Timecode Functions

Function Name	Argument(s)	Outcome

#addToCollection	*(collection, target)*
	• *collection:* The collection to modify
	• *target:* The asset to be added to the collection
	○ Can be video, clip, audio, image, document, other, project
	○ Cannot be a collection
	Example
	Expression:
	`#addToCollection(collection, video)`
	Context Data Defs:
	`<contextDataDef name="collection" dataType="Data Object" multiple="false"/>`
	`<contextDataDef name="video" dataType="Data Object" multiple="false"/>`
		Returns the provided collection.
		If the item does not already exist in the collection, it will be added.
		If the item already exists in the collection, the collection will not be modified.

#assetService	() This function takes no argument.	Returns the Asset Service object, which can be used to perform a number of asset-related tasks including creating, updating, deleting, and locking/unlocking Assets.

#collectionContents	*(collection, contentType)*
	• *collection:* The target collection. Valid input:
	  ○ Collection Data Object
	  ○ Collection ID
	  ○ Collection UUID
	• *contentType:* The type of items to return. Valid types:
	  ○ video: Timeline
	  ○ clip: Clip
	  ○ audio: AudioAssetMaster
	  ○ image: ImageAssetMaster
	  ○ document: DocumentAssetMaster
	  ○ other: AssetMaster
	  ○ project: Project
	Example
	Expression:
	`#collectionContents(collection, 'timeline')`
	`#collectionContents(collection, video.class.simpleName)`
	Context Data Defs:
	`<contextDataDef name="collection" dataType="Data Object" multiple="false"/>`
	`<contextDataDef name="video" dataType="Data Object" multiple="false"/>`
		Returns contents from the given collection matching the requested type.

#contentTemplate	*(name)*
	• *name:* The name of the content template to fetch
	Example
	Expression:
	`#contentTemplate('proxy')`
	Context Data Defs:
	`<contextDataDef name="assetContentTemplate" dataType="Data Object" multiple="false"/>`
		Returns an AssetContentTemplate for the given name.
		• Content templates have been phased out as of Studio 1.2, in favor of content uses.

#setAssetVersion	*(assetMaster, assetVersion)*
	• *assetMaster:* The asset master to update
	• *assetVersion:* The asset version to set
	Example
	Expression:
	`#setAssetVersion(assetMaster, assetVersion)`
	Context Data Defs:
	`<contextDataDef name="assetMaster" dataType="Data Object" multiple="false"/>`
	`<contextDataDef name="assetVersion" dataType="Data Object" multiple="false"/>`
		• The given *assetVersion* must already be a version of the *assetMaster*.
		• Neither *assetMaster* nor *assetVersioncan* be nulll.
		• The *assetMaster*'s current version will be updated to the given *assetVersion*.
		• If the given *assetVersion* is already the current version of the given *assetMaster*, nothing changes.

Asset Functions

Nimbus Only

The remaining functions are only available on Nimbus-based products.

Function Name	Argument(s)	Outcome
#assetContent	*(asset, contentType)* • *asset:* A string, integer, or DataObject that defines the target Asset. • *contentType:* The string, integer, or DataObject that defines the name, use, target extension, MIME type, content template ID, or object of the specific content to return from the asset. To return originally imported content, pass "null".	Returns the Asset content matching the specified type. The type can be specified in many forms.
#mezzanineContent	*(assetOrFile)* The string, file, or DataObject for the target Asset. A path or file may also be passed in, but it must be a valid Reach Engine repository file.	Returns the mezzanine Asset Content of the specified asset or asset file, if there is any.
#mezzanineTemplate	*(asset)* The string, file, or DataObject for the target Asset.	If asset is provided, returns the mezzanine Content Template associated with it. Otherwise, returns the configured default mezzanine Content Template.
#proxyContent	*(assetOrFile)* The string, file, or DataObject for the target Asset. A path or file may also be passed in, but it must be a valid Reach Engine repository file.	Returns the streaming proxy Asset Content of the specified asset or asset file, if there is any.
#proxyTemplate	*(asset)* The string, file, or DataObject for the target Asset.	If *asset* is provided, returns the streaming proxy Content Template associated with it. Otherwise, returns the configured default streaming proxy Content Template.
#thumbnailContent	*(assetOrFile)* The string, file, or DataObject for the target Asset. A path or file may also be passed in, but it must be a valid Reach Engine repository file.	Returns the thumbnail Asset Content of the specified asset or asset file, if there is any.
#thumbnailMovContent	*(assetOrFile)* The string, file, or DataObject for the target Asset. A path or file may also be passed in, but it must be a valid Reach Engine repository file.	Returns the thumbnail MOV Asset Content of the specified asset or asset file, if there is any.

#thumbnailMovTemplate	*(assetOrFile)* The string, file, or DataObject for the target Asset. A path or file may also be passed in, but it must be a valid Reach Engine repository file.	If asset is provided, returns the thumbnail MOV Content Template associated with it. Otherwise, returns the configured default thumbnail MOV Content Template.